Testing Machine Learning Code using Polyhedral Region

Md Sohel Ahmed
National Institute of Informatics
Tokyo, Japan
sohel@nii.ac.jp

Fuyuki Ishikawa
National Institute of Informatics
Tokyo, Japan
f-ishikawa@nii.ac.jp

Mahito Sugiyama
National Institute of Informatics
Tokyo, Japan
JST, PRESTO
mahito@nii.ac.jp

ABSTRACT

To date, although machine learning has been successful in various practical applications, generic methods of testing machine learning code have not been established yet. Here we present a new approach to test machine learning code using the possible input region obtained as a *polyhedron*. If an ML system generates different output for multiple input in the polyhedron, it is ensured that there exists a bug in the code. This property is known as one of theoretical fundamentals in statistical inference, for example, sparse regression models such as the lasso, and a wide range of machine learning algorithms satisfy this polyhedral condition, to which our testing procedure can be applied. We empirically show that the existence of bugs in lasso code can be effectively detected by our method in the *mutation testing* framework.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Machine learning code, Testing, Mutation Analysis, Polyhedral region, Lasso

ACM Reference Format:
Md Sohel Ahmed, Fuyuki Ishikawa, and Mahito Sugiyama. 2020. Testing Machine Learning Code using Polyhedral Region. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3368089.3417043

1 INTRODUCTION

Machine Learning (ML) algorithms have been applied in a wide range of application domains in society, and software implementations of ML algorithms have become found everywhere. With the increasing number of essential applications of ML systems, their reliability is also becoming increasingly important as failure of ML systems may cause serious problems, for example, a car accident of an automatic driving car. A fundamental step for ensuring the reliability is to establish a testing procedure that requires understanding the features and characteristics of bugs occurred. Although a number of studies have investigated bugs and fixes in various software systems, it is still challenging to perform testing in ML software due to the lack of its theoretical investigation [16]. In particular, testing ML software is fundamentally difficult as its behavior depends on training data [13].

Our goal in this paper is to present a method that can rigorously test ML code. The key to our approach is to use the theoretical bounding of the possible input region for an ML system conditioned on its output, which enables us to test ML code. Our approach can be applied to *any* ML implementation in which the behaviour of the ML algorithm can be characterized by the *polyhedral region*, which has been originally developed as the polyhedral lemma [9] to generate a probability distribution of predictions to perform unbiased statistical hypothesis testing in the framework of selective inference [15]. Whether or not the polyhedral region exists is an intrinsic property of ML algorithms, and it is already known that the region exists for a wide range of ML algorithms, for example, the lasso [17] including the higher-order sparse models [14], change-point detection [4], and GANs (generative adversarial networks) [18], to name a few. Moreover, although the polyhedral condition itself is a linear constraint, this can be applied to nonlinear methods using kernels [19]. Characterization of the polyhedral region is actively studied and its application is still growing. Therefore, the potential applicability of our testing approach is broad.

Each polyhedron is described by the pair of input and output of the ML system, and it guarantees that the output should be exactly the same if the input is located in the polyhedron. This property leads to the following observation: If a target ML system generates different output for multiple input in the same polyhedron, we can ensure that some bug must exist in its implementation. Therefore, by applying this constraint as the test assertion, we can detect code bugs if at least one example violates to the polyhedral condition. Our approach is currently limited to the system in which output is discrete; extending it to continuous output is an interesting future direction.

To show the effectiveness of our approach in ML code testing, we consider the *feature selection* scenario. In feature selection, which is one of central tasks in ML, the goal is to select relevant variables (features) from a supervised multivariate dataset; that is, given a pair $(\mathbf{X}, \mathbf{y})$ observed from a system of interest, where $\mathbf{X} \in \mathbb{R}^{n \times p}$ is a design matrix with n observations and p variables and $\mathbf{y} \in \mathbb{R}^n$ is a response vector, the task is to find a set of variables $M \subseteq \{1, \dots, p\}$ that is informative to predict $\mathbf{y}$ from $\mathbf{X}$ [7]. Hence, from the ML system viewpoint, $(\mathbf{X}, \mathbf{y})$ is input and M is output. The polyhedral region for a possible output $\mathbf{y}' \in \mathbb{R}^n$ is obtained in the form of $\mathbf{A}\mathbf{y}' \leq \mathbf{b}$ for some $\mathbf{A} \in \mathbb{R}^{n \times p}$ and $\mathbf{b} \in \mathbb{R}^p$, where $\mathbf{A}$ and $\mathbf{b}$ are computed from $\mathbf{X}$, $\mathbf{y}$, and M. This guarantees that, for any response

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7043-1/20/11.
https://doi.org/10.1145/3368089.3417043

vector $\mathbf{y}' \in \mathbb{R}^n$ that are in the polyhedron with satisfying $\mathbf{A}\mathbf{y}' \leq \mathbf{b}$, selected variables M' from $(\mathbf{X}, \mathbf{y}')$ should be exactly the same with M. Using this property, we construct a testing method for the lasso code, and empirically show that most of mutants can be successfully detected by our approach in *mutation testing* [8].

2 RELATED WORK

The most relevant work to this paper is [13], which studies testing of ML implementation. This work tackles the oracle problem of ML implementation code by multiple-implementation testing. However, [13] requires multiple implementation of the same ML algorithm as it is based on the majority voting strategy, and it is still an open problem how to test ML code from a single implementation. In summary, our work is the first to provide the way for strongly asserting the input-output relationship by incorporating a significant and evolving line of ML theories, i.e., polyhedral region.

There has been active effort on ML testing, including testing implementation of ML algorithms [2, 11]. One of the core challenges in ML testing is the oracle problem. That is, it is often impossible or too costly to define expected output for each test input. As a result, it is impossible to distinguish "incorrect" (failed) test cases that suggest existence of bugs. This point is especially true for testing implementation of ML algorithms as we run the algorithms to obtain unknown knowledge, e.g., feature selection and inference models. The oracle problem has been investigated primarily in two directions.

One is multiple-implementation testing (N-version programming). As mentioned above, it has been applied to supervised learning software [13], which derives a test input's surrogate oracle from the majority-voted output by running multiple implementations of the same algorithm. Two popular supervised learning algorithms: the k nearest neighbor classifier and the Naive Bayes classifier has been shown to be effective in recognizing faults in real-world supervised learning software. Multiple-implementation testing has been also applied to testing of inference models as well, e.g., [12]. However, it is costly to construct multiple versions of the test target. In addition, difference from other versions does not necessary mean existence of bugs as there is no unique right answer.

The other approach is metamorphic testing, e.g., [5, 10]. Instead of directly asserting the output, metamorphic testing checks whether the output changes in an expected way when we apply a certain transformation to the input value. However, this kind of relations (metamorphic relations) is difficult to find and its effectiveness is difficult to assess. For example, rerunning training by using training data revised by swapping RGB channels [5] takes some long time and it is difficult for testers to assess how effective this kind of "strange" transformations.

Differently from those studies, we propose to use constraints that directly asserts the input-output relationship, thus aiming at strong capability of bug detection [21].

3 THE PROPOSED METHOD

Here we present our ML code testing method. First we introduce the generic formulation of our testing approach, which can be applied to any implementations of ML algorithms that satisfies the polyhedral condition, which we will provide in Equation (2). Next we discuss the specific case of the lasso algorithm.

Algorithm 1 Random testing of ML code with polyhedral region

```
Require: ML code, $r_{\text{outer}}$, $r_{\text{inner}}$
Ensure: "bug exists" or "bug is not detected"
1:  for $i$ in 1 to $r_{\text{outer}}$ do
2:      Randomly generate $\mathbf{X}$, $\mathbf{y}$, and parameter $\boldsymbol{\lambda}$
3:      Compute $\mathrm{ML}(\mathbf{X}, \mathbf{y}, \boldsymbol{\lambda})$
4:      Compute $\mathbf{A}$ and $\mathbf{b}$ in Equation (1)
5:      for $j$ in 1 to $r_{\text{inner}}$ do
6:          Randomly generate $\mathbf{y}' \in \mathbb{R}^n$ such that $\mathbf{A}\mathbf{y}' \leq \mathbf{b}$
7:          Compute $\mathrm{ML}(\mathbf{X}, \mathbf{y}', \boldsymbol{\lambda})$
8:          if $\mathrm{ML}(\mathbf{X}, \mathbf{y}, \boldsymbol{\lambda}) \neq \mathrm{ML}(\mathbf{X}, \mathbf{y}', \boldsymbol{\lambda})$ then
9:              Terminate and output "bug exists"
10: Output "bug is not detected"
```

3.1 Generic Formulation

We formulate our approach in the generic case. Let $(\mathbf{X}, \mathbf{y})$ be a pair of an input matrix and an output vector of a system of interest, which forms "input" to an ML system, and let us denote by $\mathrm{ML}(\mathbf{X}, \mathbf{y}, \boldsymbol{\lambda})$ "output" of the ML system given parameters $\boldsymbol{\lambda}$. The only assumption to use our approach is that the possible region of any outcome $\mathbf{y}'$ is given in the form of

$$\mathbf{A}\mathbf{y}' \leq \mathbf{b} \tag{1}$$

with a matrix $\mathbf{A}$ and a vector $\mathbf{b}$ fully determined by $\mathbf{X}$, $\mathbf{y}$, and $\mathrm{ML}(\mathbf{X}, \mathbf{y}, \boldsymbol{\lambda})$. The inequality $\mathbf{A}\mathbf{y} < \mathbf{b}$ is interpreted as element wise, yielding a convex *polyhedron*. This polyhedral characterization is the core theoretical property to our approach. It has been originally developed to address the problem of *selective inference*, which is for creating accurate uncertainty evaluations for the parameters estimated by a feature selection algorithm such as the lasso [17]. The seminal paper by Lee et al. [9] provides the *polyhedral lemma*, which gives us a probability distribution over the space of the polyhedron and it can be used to remove a bias in statistical hypothesis testing [20]. We therefore obtain the *polyhedral condition* as

$$\left\{ \mathbf{y}' \;\middle|\; \mathrm{ML}(\mathbf{X}, \mathbf{y}, \boldsymbol{\lambda}) = \mathrm{ML}(\mathbf{X}, \mathbf{y}', \boldsymbol{\lambda}) \right\} = \left\{ \mathbf{y}' \;\middle|\; \mathbf{A}\mathbf{y}' \leq \mathbf{b} \right\}. \tag{2}$$

In our testing method, we first prepare an example pair $(\mathbf{X}, \mathbf{y})$, which can be randomly created, and apply the ML system to obtain its result $\mathrm{ML}(\mathbf{X}, \mathbf{y}, \boldsymbol{\lambda})$ and $\mathbf{A}$, $\mathbf{b}$ in Equation (1). Then, for any $\mathbf{y}'$ satisfying the condition $\mathbf{A}\mathbf{y}' \leq \mathbf{b}$, the polyhedral condition in Equation (2) tells us that

$$\mathrm{ML}(\mathbf{X}, \mathbf{y}, \boldsymbol{\lambda}) = \mathrm{ML}(\mathbf{X}, \mathbf{y}', \boldsymbol{\lambda}) \tag{3}$$

should be always satisfied. Therefore, if we can find at least one $\mathbf{y}'$ that does not satisfy the above condition (3), we can surely say that the target ML code includes a bug. In contrast, if the code does not have any bug, the condition (3) always holds.

The pseudo-code of our testing method is shown in Algorithm 1. We repeat the above procedure for many randomly generated pairs $(\mathbf{X}, \mathbf{y})$ and test output $\mathbf{y}'$, and try to find $\mathbf{y}'$ that violates the condition (3). Note that, if at least $\mathbf{y}'$ is found, then our method ensures that the code includes a bug and immediately terminates. Note that the presented method is simple random testing in terms of input generation. One can deploy sophisticated input generation techniques such as adaptive random testing [3] or search-based testing [1].

3.2 Specific Application for Lasso

We apply our method to the case of the lasso [17], as it is the original application of the polyhedral characterization and its properties have been well established. The *lasso* (least absolute shrinkage and selection operator) is one of the most popular methods for sparse regression, and it can perform variable (feature) selection through learning of a linear regression model. Given a design matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$ with n observations with p variables and its response vector $\mathbf{y} \in \mathbb{R}^p$, the problem of the lasso is formulated as finding the optimal coefficients $\hat{\boldsymbol{\beta}} \in \mathbb{R}^p$ such that

$$\hat{\boldsymbol{\beta}} \in \operatorname*{argmin}_{\boldsymbol{\beta} \in \mathbb{R}^p} \frac{1}{2} \|\mathrm{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda \|\boldsymbol{\beta}\|_1, \tag{4}$$

where $\|\cdot\|_1$ and $\|\cdot\|_2$ are L^1 and L^2 norms, respectively, given as $\|\boldsymbol{x}\|_1 = |x_1| + |x_2| + \cdots + |x_p|$ and $\|\boldsymbol{x}\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_p^2}$. By solving the above optimization problem, we can select a set of variables $\hat{M} = \{j \in [p] \mid \hat{\beta}_j \neq 0\}$ with $[p] = \{1, 2, \dots, p\}$ from p variables (features). This set $\hat{M}$ is called the *active set* and is considered to be the set of selected variables by the lasso. Here we provide analytical solution of $\mathbf{A} \in \mathbb{R}^{n \times p}$ and $\mathbf{b} \in \mathbb{R}^n$ in Equation (1), which as been presented in [9, Proposition 4.2]. Let $(\mathbf{Z}^T\mathbf{Z})^+$ be the (Moore-Penrose) pseudoinverse of the square matrix $\mathbf{Z}^T\mathbf{Z}$ and $\mathbf{Z}^+ = (\mathbf{Z}^T\mathbf{Z})^+\mathbf{Z}^T$, $\mathbf{X}_M \in \mathbb{R}^{n \times |M|}$ be the submatrix of $\mathbf{X}$ with respect to columns of M, $\mathbf{X}_{-M} \in \mathbb{R}^{n \times (n - |M|)}$ be the submatrix with respect to columns of $[p] \setminus M$, and $\mathbf{P}_M = \mathbf{X}_M(\mathbf{X}_M^T\mathbf{X}_M)^{-1}\mathbf{X}_M$ be projection onto the columns space of $\mathbf{X}_M$. Then if the lasso chooses a variable set $\hat{M}$ with its sign vector $\hat{\mathbf{s}}$; that is, $\mathrm{ML}(\mathbf{X}, \mathbf{y}, \lambda) = (\hat{M}, \hat{\mathbf{s}})$, we have

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_0(\hat{M}, \hat{\mathbf{s}}) \\ \mathbf{A}_1(\hat{M}, \hat{\mathbf{s}}) \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} \mathbf{b}_0(\hat{M}, \hat{\mathbf{s}}) \\ \mathbf{b}_1(\hat{M}, \hat{\mathbf{s}}) \end{pmatrix}, \tag{5}$$

where

$$\mathbf{A}_0(\hat{M}, \hat{\mathbf{s}}) = \frac{1}{\lambda} \begin{pmatrix} \mathbf{X}_{-\hat{M}}^T (\mathbf{I} - \mathbf{P}_{\hat{M}}) \\ -\mathbf{X}_{-\hat{M}}^T (\mathbf{I} - \mathbf{P}_{\hat{M}}) \end{pmatrix}, \ \mathbf{b}_0(\hat{M}, \hat{\mathbf{s}}) = \begin{pmatrix} \mathbf{1} - \mathbf{X}_{-\hat{M}}^T (\mathbf{X}_{\hat{M}}^T)^+ \hat{\mathbf{s}} \\ \mathbf{1} + \mathbf{X}_{-\hat{M}}^T (\mathbf{X}_{\hat{M}}^T)^+ \hat{\mathbf{s}} \end{pmatrix},$$

$$\mathbf{A}_1(\hat{M}, \hat{\mathbf{s}}) = -\mathrm{diag}(\hat{\mathbf{s}})(\mathbf{X}_{\hat{M}}^T\mathbf{X}_{\hat{M}})^{-1}\mathbf{X}_{\hat{M}}^T, \quad \text{and}$$

$$\mathbf{b}_1(\hat{M}, \hat{\mathbf{s}}) = -\lambda \mathrm{diag}(\hat{\mathbf{s}})(\mathbf{X}_{\hat{M}}^T\mathbf{X}_{\hat{M}})^{-1}\hat{\mathbf{s}}.$$

In the above equations, $\mathbf{A}_0$ and $\mathbf{b}_0$ encode the "inactive" constraints, and $\mathbf{A}_1$ and $\mathbf{b}_1$ encode the "active" constraints. We illustrate the polyhedral region in the lasso in Figure 1.

4 EXPERIMENTS

We examine the effectiveness of our method using mutation testing, which evaluates the quality of a testing method by counting how many mutants (code including bugs) are detected by the method. All experiments were conducted in Cent OS release 6.10 on a single core 2.20 GHz Intel Xeon CPU E7-8880 v4 and 2 TB of memory. We implemented our method on the R language version 3.5.2.

To perform mutations testing, we used R implementation of the total 142 lines of lasso code, the function `glmnet` provided in the glmnet library [6], which is treated as the "correct" lasso code. Based on the code, we generated 177 mutants of the lasso code in total using four types of basic mutations including arithmetic mutation, relational mutation, logical mutation, and assignment mutation. The list of mutations is summarized in Table 1. In our method, we set $n = 50$, $p = 10$, $\lambda = 0.8$, $r_{\mathrm{outer}} = r_{\mathrm{inner}} = 1{,}000$. That is, in the outer loop starting from line 1 in Algorithm 1, we repeat randomly generating $\mathbf{X} \in \mathbb{R}^{50 \times 10}$ and $\mathbf{y} \in \mathbb{R}^{50}$ 1,000 times, and for each dataset, we also repeat 1,000 times generating $\mathbf{y'} \in \mathbb{R}^{50}$ in the inner loop starting from line 5. If we detect one $\mathbf{y'}$ that satisfies the condition in line 8, we terminate the algorithm and output "bug exists", where it ensures that a bug exists in the code and our method can successfully kill the mutant. Otherwise any $\mathbf{y'}$ does not satisfy the condition, we output "bug is not detected", which means that our method fails to detect the mutant.

Results are summarized in the rightmost two columns in Table 1. 45 mutants were detected by the compiler error of R. In the rest of 132 mutants, our method successfully detected 129 mutants. Therefore, our method successfully detected 98 % (129/132) of mutants, and only three mutants were not detected. Interestingly, all three survived mutants are generated by mutating the "less than" operation. Our methods required 69.51 ± 16.70 inputs to detect each mutant, which took about 10 seconds and is reasonably efficient to perform mutation testing. Our results therefore show that our method can effectively and efficiently detect various types of mutants of the lasso code. Since our method depends on randomly generated datasets, it is important to try many input to detect mutants in a systematic manner. Currently, we tested 1,000 randomly generated input pairs, thus it may be possible to find more mutants with more input. It is our interesting future work to design an approach that generates $(\mathbf{X}, \mathbf{y})$ and $\mathbf{y'}$ in an efficient manner.

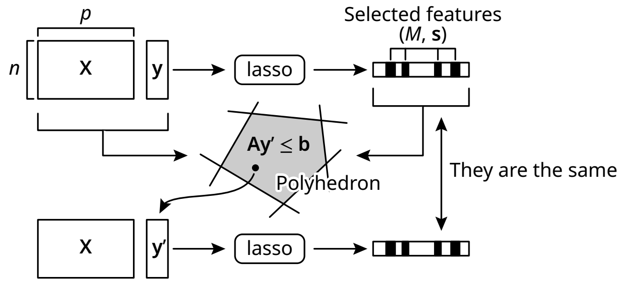

Figure 1: Polyhedral region in lasso.

5 CONCLUSION

In this paper, we have proposed a testing approach for ML code based on the polyhedral region of possible input conditioned on output. Our empirical evaluation shows that our testing method successfully detects 98 % of mutants of the lasso code efficiently, which is currently impossible by any other testing methods. Since this polyhedral characterization applies to not only the lasso but a broad range of ML algorithms, our method has a wide applicability for the ML code testing. Our method can be a fundamental of ML testing techniques, and will lead to further development of testing methods for ML implementations.

Table 1: Generated mutants from lasso code and experimental results.

	Mutants			Experimental results	
Mutation operators	Target operators	# of mutated places	# of mutants	# detection by our method	# detection by compiler
Arithmetic mutation	plus	4	20	20	0
	minus	3	15	6	9
	multiplication	1	5	5	0
	division	0	0	0	0
	modulus	0	0	0	0
	exponentiation	0	0	0	0
Relational mutation	equal	10	50	38	12
	not equal	3	15	12	3
	less than or equal to	1	5	2	3
	greater than or equal to	1	5	2	3
	greater than	4	20	17	3
	less than	7	35	20	12
Logical mutation	or	1	3	3	0
	and	0	0	0	0
	or2	0	0	0	0
	and2	1	3	3	0
Assignment Mutation	left assignment	1	1	1	0
	right assignment	0	0	0	0
Total		37	177	129	45

ACKNOWLEDGMENTS

This work was supported by JST MIRAI Grant No. JPMJMI18BB; Council for Science, Technology and Innovation (CSTI), Cross-ministerial Strategic Innovation Promotion Program (SIP), "Materials Integration for revolutionary design system of structural materials" (Funding agency: JST; SA and MS); and JST, PRESTO Grant No. JPMJPR1855, Japan (MS).

REFERENCES

[1] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. 2010. A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation. *IEEE Transactions on Software Engineering* 36, 6 (2010), 742–762.

[2] Houssem Ben Braiek and Foutse Khomh. 2020. On testing machine learning programs. *Journal of Systems and Software* 164 (2020), 110542.

[3] T. Y. Chen, H. Leung, and I. K. Mak. 2005. Adaptive Random Testing. In *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, Michael J. Maher (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 320–329.

[4] V. N. L. Duy, H. Toda, R. Sugiyama, and I. Takeuchi. 2020. Computing Valid p-value for Optimal Changepoint by Selective Inference using Dynamic Programming. *arXiv:2002.09132* (2020).

[5] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, R. P. Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *The 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018).* 118–120.

[6] J. Friedman, T. Hastie, and R. Tibshirani. 2010. Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of statistical software* 33, 1 (2010), 1–22.

[7] I. Guyon and A. Elisseeff. 2003. An introduction to variable and feature selection. *Journal of Machine Learning Research* 3 (2003), 1157–1182.

[8] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering* 37, 5 (2010), 649–678.

[9] Jason D. Lee, Dennis L. Sun, Yuekai Sun, and Jonathan E. Taylor. 2016. Exact post-selection inference, with application to the lasso. *The Annals of Statistics* 44, 3 (2016), 907–927.

[10] Christian Murphy and Gail E Kaiser. 2011. Improving the dependability of machine learning applications. *Columbia University Computer Science Technical Reports* (2011).

[11] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proceedings of the 36th International Conference on Machine Learning*, Vol. 97. 4901–4911.

[12] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *The 26th Symposium on Operating Systems Principles (SOSP 2017).* 1–18.

[13] Siwakorn Srisakaokul, Zhengkai Wu, Angello Astorga, Oreoluwa Alebiosu, and Tao Xie. 2018. Multiple-implementation testing of supervised learning software. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence.*

[14] S. Suzumura, K. Nakagawa, Y. Umezu, K. Tsuda, and I. Takeuchi. 2017. Selective Inference for Sparse High-Order Interaction Models. In *Proceedings of the 34th International Conference on Machine Learning*, Vol. 70. 3338–3347.

[15] Jonathan Taylor and Robert J Tibshirani. 2015. Statistical learning and selective inference. *Proceedings of the National Academy of Sciences* 112, 25 (2015), 7629–7634.

[16] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering.* 271–280.

[17] Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* 58, 1 (1996), 267–288.

[18] Y.-H. H. Tsai, D. Wu, M. Yamada, R. Salakhutdinov, I. Takeuchi, and K. Fukumizu. 2018. Selecting the Best in GANs Family: a Post Selection Inference Framework. In *Workshop Track Proceedings of the 6th International Conference on Learning Representations.* 1–4.

[19] M. Yamada, Y. Umezu, K. Fukumizu, and I. Takeuchi. 2018. Post Selection Inference with Kernels. In *Proceedings of the 21st International Conference on Artificial Intelligence and Statistics*, A. Storkey and F. Perez-Cruz (Eds.), Vol. 84. 152–160.

[20] Fan Yang, Rina Foygel Barber, Prateek Jain, and John Lafferty. 2016. Selective inference for group-sparse linear models. In *Advances in Neural Information Processing Systems.* 2469–2477.

[21] Yucheng Zhang and Ali Mesbah. 2015. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* ACM, 214–224.